Science of Computer Programming 75 (2010) 921–942

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A novel analysis space for pointer analysis and its application for bug finding

Marcio Buss [a,b,*], Daniel Brand [b], Vugranam Sreedhar [c], Stephen A. Edwards [a]

[a] Department of Computer Science, Columbia University, New York, NY, United States
[b] IBM T. J. Watson Research Center, Yorktown Heights, NY, United States
[c] IBM T. J. Watson Research Center, Hawthorne, NY, United States

ARTICLE INFO

Article history:
Received 30 July 2008
Received in revised form 30 March 2009
Accepted 12 August 2009
Available online 26 September 2009

Keywords:
Static analysis
Pointer analysis
Summary-based analysis
Bug-finding

ABSTRACT

The size of today's programs continues to grow, as does the number of bugs they contain. Testing alone is rarely able to flush out all bugs, and many lurk in difficult-to-test corner cases. An important alternative is static analysis, in which correctness properties of a program are checked without running it. While it cannot catch all errors, static analysis can catch many subtle problems that testing would miss.

We propose a new space of abstractions for pointer analysis—an important component of static analysis for C and similar languages. We identify two main components of any abstraction—how to model statement order and how to model conditionals, then present a new model of programs that enables us to explore different abstractions in this space. Our assign-fetch graph represents reads and writes to memory instead of traditional points-to relations and leads to concise function summaries that can be used in any context. Its flexibility supports many new analysis techniques with different trade-offs between precision and speed.

We present the details of our abstraction space, explain where existing algorithms fit, describe a variety of new analysis algorithms based on our assign-fetch graphs, and finally present experimental results that show our flow-aware abstraction for statement ordering both runs faster and produces more precise results than traditional flow-insensitive analysis.

1. Introduction

Modern society is irreversibly dependent on computers and, consequently, on software. Examples abound: from office work to banking, from leisure to air traffic control, communications, cars, air planes, etc. Yet, software products today are plagued by defects. The increasing complexity of programs makes it almost impossible to deploy an error-free product. For example, the Linux kernel grew from 2 million lines of code in 2002 to about 6 million lines in 2007 [1]; the number of lines of code in a typical GM vehicle increased from 100 thousand in 1970 to 1 million in 1990—it is estimated to grow to 100 million lines by 2010 [2]. As a striking example, the Windows 2k operating system was shipped with 63,000 defects (discovered to date) [3]. Furthermore, it has been argued from empirical evidence [4] that even bug *density* (i.e., the number of defects per lines of code) is increasing with project size.

Static program analysis has been gaining renewed momentum as a solution for automated error detection. In the most basic sense, it means finding defects without running the code. More specifically, it means analyzing the source code and

* Corresponding author at: Department of Computer Science, Columbia University, New York, NY, United States.
E-mail address: marcio@cs.columbia.edu (M. Buss).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.08.002

searching for violations of correctness properties. This approach is complementary, and sometimes more attractive, to more traditional methods such as *testing* [5] (dynamic analysis) or *model checking* [6,7].

However, static analysis invariably issues some false alarms because it considers all paths through the program simultaneously and symbolically executes the program instead of using real inputs. Thus, static tools will inevitably signal errors in correct programs. By contrast, dynamic analysis examines particular runs of a program at a finer granularity and thus never issues false alarms because it has precise information about the current execution state. Thus, a static bug finder tends to issue either too many spurious warnings about non-bugs, or is too uncertain about program defects and thus misses real bugs.

The presence of pointers is a challenging aspect of many languages that can force static analysis to consider far more behavior than actually possible. In C, pointers are often regarded as the bane of static program analysis. They pose a problem to compilers and bug finding tools because it is often unclear what locations may be accessed through indirect memory references. Aliasing, i.e., two expressions referring to the same memory location, is another common aspect of using pointers, which can complicate the analysis. Without enough information, static tools are forced to make conservative assumptions such as "a pointer assignment could write to any variable in the current scope", leading to many superfluous dependencies and significantly limiting the power of the analyzer.

To alleviate that, a common solution is to allow the programmer to provide annotations to guide the analyzer, e.g., to declare a variable as unaliased in a certain scope, allowing the analyzer to aggressively infer properties in that scope without fearing any aliases. Other tools simply restrict pointer use to a minimal, systematic way [8], provide special constructs such as the "never-null" pointer of Cyclone [9], or create special data types that can never be aliased [10].

However, experience has shown [11] that programmers are reluctant to provide any but the most minimal annotations, and when they do, the annotations are rarely synchronized with code changes. This burden is particularly pronounced when applying a tool to a legacy code base.

The solution is to provide some external checking of properties. For pointers, determining useful information requires some form of *pointer analysis*. Such analysis consists of computing *points-to* information—given two program locations, p and q, we say p points-to q if p can contain the address of q. Pointer analysis statically estimates such possible set of locations a pointer can point to during program's execution. There are many types of pointer analysis, with different levels of precision and speed. The precision of a particular analysis can directly affect the utility of the bug finding tool—the more precise the pointer analysis, the more aggressive the tool can be, leading to fewer false alarms and/or missed errors. However, some solutions are precise but prohibitively expensive, while others are fairly cheap but too approximate. A bug finder that relied on the latter would be too inaccurate, while using the former could result in a checker that is unacceptably slow. The gap between these two extremes is sparsely populated by a few disconnected, ad-hoc heuristics. This has prevented designers from implementing effective pointer analysis algorithms into real-world error detection tools, even though researchers agree [12,13] that this could lead to considerable benefits.

We propose a novel analysis space for pointer analysis in which the usual extreme solutions are simply special cases of a more fundamental, underlying principle. This is achieved by reformulating the granularity and the dimensions of pointer analysis itself, allowing finer-grain trade-offs between speed and utility, as well as finding new, previously unknown, sweet spots. In particular, we introduce flow-aware analysis—a technique that approximates statement ordering information and is thus more precise than flow-insensitive analysis, yet runs faster since it processes far fewer alias relationships because the increased precision reduces the number that need to be considered.

The framework is particularly tailored for what we call modular bug finding [14], and it is developed around a new abstraction for computing points-to sets, the Assign-Fetch Graph, that has many interesting features. Empirical evaluation shows interesting results, as some unknown errors in well-known applications (such as the Linux kernel) were discovered.

While our pointer analysis technique is targeted to bug finding, its results are never directly reported to the user because they are still too approximate. Instead, they are used to provide hints to cut down our reliance on a much more accurate, but vastly more expensive, analysis procedure. We discuss this more when we report experimental results in Section 10.

This work is a distillation of the first author's thesis [14], which we also presented in much-abbreviated form elsewhere [15].

2. Pointer analysis

Pointer analysis attempts to statically determine useful properties about pointers in a program. Most interesting properties are undecidable, so most algorithms strive for a sound approximation. In this paper, we are concerned with *points-to analysis* [16], which, for each pointer, determines a superset of the locations to which it can refer; and *alias analysis* [17], which calculates pairs of pointer expressions that may refer to the same storage location. We do not consider *escape analysis* [18–20], which identifies which memory locations may be visible outside a particular scope; and *shape analysis* [21,22], which tries to understand the structure of data in a program, e.g., whether something is a tree.

Points-to analysis algorithms are distinguished by how they abstract the program. *Flow-sensitive analysis* [23–25,16,17] considers the order in which the statements of a program execute, and is usually based on an iterative dataflow framework [26]. More common is a *flow-insensitive analysis* [27–32], which treats the program as a pile of (unordered) statements, much like a type system. As such, flow-insensitive algorithms are generally formulated as a set of type rules.

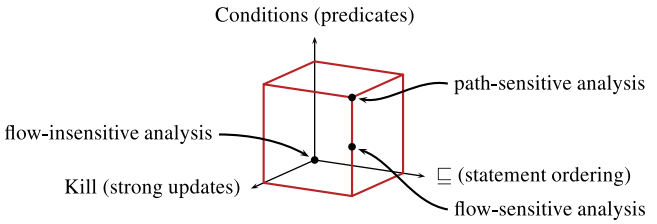


Fig. 1. Our abstraction model. Each point in this three-dimensional space corresponds to a different interpretation of a program; we indicate three that correspond to common pointer analysis algorithms. Moving away from the origin means modeling behavior more precisely, but not necessarily a more expensive analysis.

Such an analysis may compute a single solution that is valid for the entire program [27,33,32,34] or may compute one solution for each function [29,35].

The flow-insensitive approach trades accuracy for speed, but it is often accurate enough. In particular, it remains sound and thus safe for applications such as compiler optimization, where imprecision merely reduces the quality of the result. A number of studies have tried to quantify the trade-offs between the two techniques [36–38].

Another dimension of these algorithms is *context-sensitivity*, which refers to how much multiple calls to the same function are differentiated. Using program state information at a function's call site yields a context-sensitive analysis.

Many context-sensitive algorithms summarize the behavior of a called function as a transfer function. The algorithm creates a parametrized summary of the behavior of a function, then uses it to characterize the behavior of its callers. While such an approach is theoretically appealing, its implementation is not straightforward, especially when the summary function cannot be expressed symbolically. *Assumption sets* [39] are a widely adopted alternative that represent the transfer function as a table of mappings from context information to output flow values. A context-sensitive analysis algorithm selects different entries from this table based on the program state at a call site.

Since not every possible state will occur at a particular call site, it is natural to consider partial transfer functions [40]—function summaries that are only computed for the input patterns encountered during a top-down traversal of a program's functions. Here, a context is the aliasing relationship among function arguments (e.g., whether the first and second arguments of a function may point to the same array).

Our technique for context-sensitivity is based on a functional approach that is not based on assumption sets. It works by representing summary information symbolically in the form of a partially evaluated graph.

The needs of evidence-based bug detection [41] motivated our form of pointer analysis. Those needs are different from those of whole program analysis. The difference can be best understood by an example: suppose a caller passes a null pointer to a callee, who then dereferences it. Is the bug in the callee for failure to check its parameter, or is the bug in the caller for passing null to a function not intended to handle it? Whole program analysis flags the error in the callee, thus whether a callee contains an error or not depends on its calling environment. In contrast, in evidence-based analysis, the presence of a bug inside a procedure never depends on its calling environment. In our example, the callee contains a bug only if there is evidence that the callee is "meant" to handle a null argument. (An example of such evidence is a separate test for it being null). In the absence of such evidence, the callee is not in error. Instead, information about its requirement for a non-null argument is propagated to any caller. If a caller violates this requirement by passing a null pointer, the caller is in error.

Thus, evidence-based bug detection implies information about functions—including pointer alias information—propagated from callees to callers, never in the opposite direction. This poses a particular challenge for pointer analysis because pointer relationships inside callees in general do depend on callers. This paper presents a solution to this problem—an accurate pointer information for a procedure is obtained by propagating information bottom-up from callee to caller, but never top-down.

3. A space of abstractions for pointer analysis algorithms

The goal of our work is to formalize and improve the flexibility of the abstractions used in pointer analysis. We do so by decomposing the abstraction problem into three dimensions: statement ordering, conditions, and strong updates, which we present in this section. Later, we show several ways to combine these elements to produce new analysis algorithms, including a new "sweet spot"—our flow-aware analysis that is both faster and more precise than flow-insensitive analysis. Exploring the analysis space is a step towards bridging the gap between cheap and approximate versus precise and expensive algorithms.

Fig. 1 shows the essence of our abstraction space. Each dimension corresponds to a particular aspect of the program's semantics (the ordering of statements, the effects of conditionals, and the treatment of assignments). We place the most abstract interpretation of a program at the origin. There is no numeric scale associated with axes and there is no implication of any total ordering—it is just a qualitative representation of the independence of the three aspects of program semantics. Each axis has two important points—the origin, where information from the program source is completely ignored, and the maximum point, where all information is used. Points in between represent partial use of program information.

The right-pointing axis ("$\sqsubseteq$") represents how precisely an algorithm heeds the ordering of program statements. Traditional flow-insensitive analysis completely ignores statement ordering (e.g., it assumes that any two statements may be executed in either order) and thus corresponds to the origin. A precise analysis would consider the exact partial ordering of statements in a program. In Section 6.1, we describe flow-aware analysis, an efficient approximation that treats statements as totally ordered.

The vertical axis ("Conditions") represents how an algorithm abstracts the effects of conditional statements such as if-then-else. Again, the origin corresponds to ignoring this aspect of program execution by assuming that statements in every branch of a conditional may all execute, again, a traditional assumption in flow-insensitive analysis. Points along this axis correspond to modeling mutual exclusion among branches, relationships among branches under distinct conditionals, and so forth. We consider a technique for modeling the effect of conditionals in Section 7.

The left-pointing axis ("Kill") refers to how strong updates are handled, i.e., whether the effect of an assignment is treated as overwriting all earlier assignments to the same memory location. While an assignment in a running program does exactly this, it becomes difficult to model when statement evaluation order, the behavior of conditionals, and the concrete values of pointers are unknown, as it is in static analysis. Naturally, varying levels of approximation are possible, e.g., updates to variables that are never aliased may be modeled exactly while pointers may be approximated. In this paper, we do not propose new abstractions for this axis; see the first author's thesis [14].

Fig. 2 illustrates how different points in our abstraction space correspond to different abstractions and produce different results. Here, we show how the small program in Fig. 2(g) is interpreted under various abstractions to produce different points-to graphs.

Fig. 2(a) is a precise analysis that considers statement order, conditionals, and strong updates and thus produces the most sparse points-to graph. For instance, the dereference of `p` at statement `*p=&w` cannot read the address of `r` because `p=&r` occurs on the opposite branch. Also, statement `p=&x` is killed by `p=&q` prior to statement `*p=&w`, and therefore `q` is the only location that can be set to point to `w`.

Fig. 2(b) is the other extreme: it ignores statement order and conditionals (we omitted the "c" node to indicate this). Not surprisingly, this produces many spurious aliases, such as *r* pointing to *w*, which can never happen in the original program. This is equivalent to an Andersen-style analysis [27].

Fig. 2(c) illustrates what happens when only conditionals are ignored. Here, we replaced the conditional node "c" with a triangle node to indicate that both branches of the conditional are treated as running in parallel. Although statement ordering is considered overall, the relative order of statements in opposite branches of the conditional is non-deterministic. This means `*p` in statement `*p=&w` can read either `q` or `r`, and thus both points-to relations `q` $\rightsquigarrow$ `w` and `r` $\rightsquigarrow$ `w` become valid. Similarly, because the right-hand side in `z=p` can read the value set by `p=&q`, statement `*z=&t` also sets `q` to point-to `t`.

Fig. 2(d) shows the effects of considering only statement ordering. Compared to Fig. 2(b), it omits the arc from *z* to *r* because although the program contains both `p=&r` and `z=p`, `p=&r` appears later so we know that *z* cannot take on the address of *r*. Fig. 2(d) also illustrates the effect of ignoring strong updates in the abstraction of Fig. 2(c). For example, the deference of `p` in `*p=&w` can also refer to the address of `x`, and make that variable to point-to `w` as well.

Fig. 2(e) adds the effect of conditionals, which inhibits the interaction between the statements in the two branches of the conditional, eliminating, for example, the points-to arc between *r* and *w*. Finally, Fig. 2(f) only heeds the effect of conditionals, but not statement ordering. This adds, for example, the *r*-to-*t* arc because the `p=&r` assignment is now considered as possibly running before `z=p`.

Choosing one of these pointer-analysis variations may impact the outcome of another analysis. Assume that a false-positive-suppressing tool wants to prove that the statement `*z=&t` does not write to *q*. Then, only the solutions in Figures 2(a), (e), and (f) would suffice; all others have a (spurious) arc from *z* to *q*.

Some of the different abstraction points in Fig. 2 have been implemented by others using very different algorithms for each point. Below, we describe one of our main contributions: the Assign-Fetch Graph (AFG), a representation for program behavior that makes it easy to implement different abstractions. Slight variations in how this graph is built and interpreted lead directly to the abstractions shown in Fig. 2 and facilitate exploring different points in our abstraction space.

4. The assign-fetch graph

Instead of the usual *points-to graph* [31,16,32] used by most pointer analysis algorithms, we represent a function's behavior with an assign-fetch graph (AFG), which we developed to make it easier to abstract the program in different ways. Our analysis is *summary-based* [42,43,40]: it computes a representation for each procedure that summarizes its effects on pointers.

Unlike a points-to graph, whose nodes represent pointer variables and whose edges represent points-to relations, the nodes in our AFG represent locations and values, and edges represent reads and writes to memory. Fig. 3 illustrates the assign-fetch graph for a small program fragment. Pointer analysis amounts to matching pointer dereferences to pointer assignments, i.e., the analysis can be thought of as an attempt to understand which writes could be seen by each read. One approximation is that each write to a location can be seen by every read of that location, but this is usually an over-approximation: a read and write may occur in different branches of a conditional, or a write might occur in sequence after a read. Our analysis space enables us to approximate these relationships in various ways, producing different pointer analysis algorithms; the AFG is the main data structure behind it.

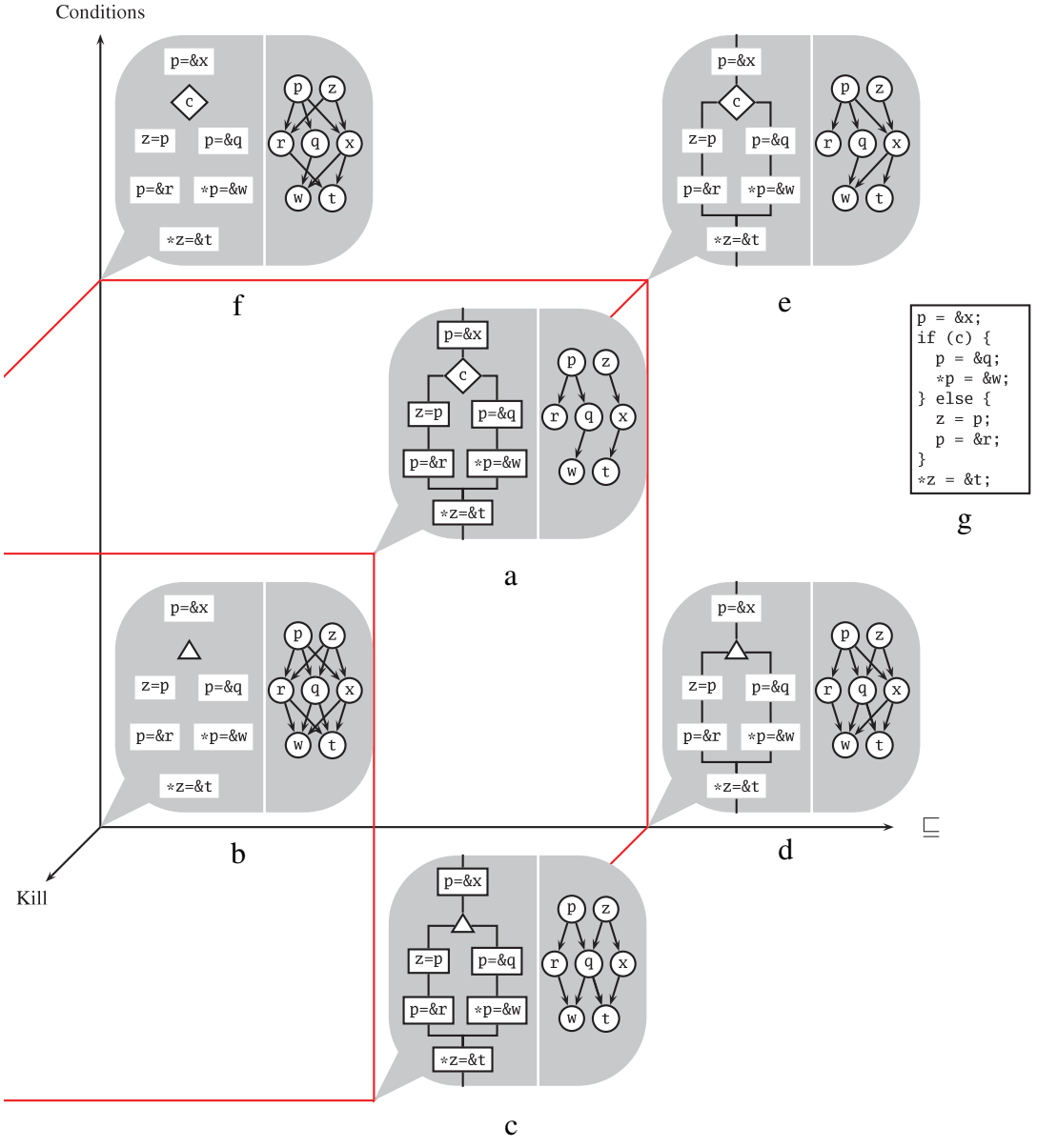


Fig. 2. Our abstraction model applied to a small program (g). Each point in this three-dimensional space corresponds to a different interpretation of a program (a)–(f), which we illustrate with a control-flow graph and the points-to graph it produces.

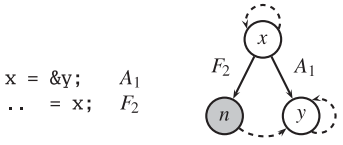


Fig. 3. A pair of statements and their AFG: x is assigned in A_1 and fetched in F_2; an alias edge indicates that n, the result of the fetch, can take the address of y.

In traditional pointer analysis, there is a major difference between flow-sensitive and flow-insensitive analysis. In flow-insensitive analysis, there is a single points-to graph representing a solution for the whole program. In contrast, a flow-sensitive analysis associates a points-to graph with each control flow point. Like flow-insensitive analysis, we maintain a single AFG solution, even in a mode as precise as flow-sensitive analysis. This is possible by having the two kinds of edges (assign and fetch), as opposed to only one kind of an edge in a points-to graph. The distinction between assigns and fetches allows us to express that one may happen before the other, or that they may happen under different conditions. This makes it unnecessary to associate different information with different program points.

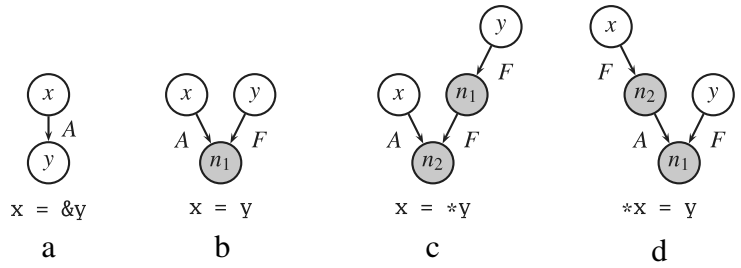


Fig. 4. The usual statements considered in pointer analysis and their AFGs.

Our total system deals with three kinds of graphs—a syntax tree, a flow graph, and the AFG. Only the last is the subject of this paper, but it is important to avoid confusion with the other two. In a syntax tree an edge goes from a node representing a syntactic object to one of its syntactic sub-objects. A flow graph is obtained from a syntax tree by adding semantic information; an edge indicates that one piece of data is used in calculating another. AFG is obtained from a flow graph and represents a state of memory; an assign or fetch edge goes from an address to its contents.

Formally, an *assign-fetch graph* is a 7-tuple (N, C, T, I, A, F, E) where N is the set of nodes, of which $C \subset N$ are fetch nodes, $T \subset N - C$ are interface nodes and $I \subset N$ are initial-value nodes ($I \subset N - C$ before summarization and $I \subset C$ after. $T \cap I = \emptyset$). $A : N \times N$ are assignment edges; $F : N \times N$ are fetch edges, and $E : N \times N$ are alias edges.

We write $n, n_1, n_2, \ldots \in C$ to denote fetch nodes, $x, y, \ldots \in N - C$ for location nodes; and $\alpha, \beta, \ldots \in N$ for arbitrary nodes. We write $\mathrm{al}(\alpha) = \{\beta : (\alpha, \beta) \in E\}$ to indicate the set of nodes to which α may refer, i.e., for which α may be an alias.

An AFG represents memory locations and values (the nodes N), and operations (loads and stores—F and A). A fetch node $n \in C$ represents a value that has been read from memory; each fetch edge, $(\gamma, n) \in F$, represents a read of location γ that produces the value n. We often write this as $\gamma \xrightarrow{F} n$. A fetch node usually has exactly one incident fetch edge. Each assignment $(\alpha, \beta) \in A$ represents a write to the location α with the value (an address) β. We write this $\alpha \xrightarrow{A} \beta$.

Interface nodes $T \subset N - C$ represent values from outside that are visible within a function: argument values, global variables, and heap locations. Local (automatic) variables are never interface nodes. Initial-value nodes I represent values taken by the interface nodes when a function is entered. Interprocedural analysis treats these nodes specially (see Section 8); they are otherwise undistinguished.

Fig. 3 shows the AFG for a simple pair of assignments to illustrate our notation. Here, the three nodes $N = \{x, y, n\}$ represent the addresses of variables x and y and "n," the result of reading the contents of x.

The first statement assigns the address of y to x. This is the only assignment in this fragment, so $A = \{(x, y)\}$, which we indicate by the arc from x to y labeled A_1.

In the second statement, we read the value of x. Considering the statement in isolation, we do not know the value of x, so we represent it with the (shaded) fetch node n ($C = \{n\}$). Since this node represents a fetch of x (and the only fetch in this fragment), $F = \{(x, n)\}$ and we draw the arc labeled F_2 from x to n.

Fetch nodes represent unknown values; the goal of pointer analysis is to resolve such unknowns. In this simple example, it is easy to see that x could only contain the address of y at this point. Thus, we say n is an alias for y. Since x and y can only refer to themselves, $E = \{(n, y), (x, x), (y, y)\}$, and we draw a dashed line from n to y and self-loops on x and y.

Because each named variable is assumed to have a unique address, each location node is an alias for itself and only itself. However, for clarity in later figures, we do not draw the alias self-loop on each location node; we simply assume it is there.

The AFG represents more information about the behavior of the program than a points-to graph (which for Fig. 3 would consist of just a single arc from x to y) to allow us to divide pointer analysis algorithms into better-understood pieces. Constructing nodes and edges for a program is a mechanical procedure. However, alias edges can be added in many ways; each technique corresponds to a different abstraction.

Fig. 4 shows AFG fragments for the four statements typically considered in pointer analysis. For `x=&y`, we represent the lvalue `x` and the rvalue `&y` as location nodes and connect them with an assign edge indicating `x` points to the memory location for `y`. Note that this does not read or change the contents of `y`. By contrast, since `x=y` reads `y`, we add the fetch node n_1 to represent its value, add a fetch edge indicating a read of `y`, and add an assign edge to indicate that they are written to `x`. We treat other statements similarly.

Fig. 5 illustrates an AFG being interpreted for different abstractions. Fig. 5(b) is the AFG constructed mechanically from the C code in Fig. 5(a). The first statement, `*z=&x`, stores the address of global variable x at the address in z. We represent z with a location node labeled z, the dereference of z with the fetch edge F_1, the address read from z with the fetch node n_1, the address of x with the location node x, and the write to the address contained in z with the assign edge A_2.

Again, a (shaded) fetch node represents the value returned by a particular fetch operation in the program. We do not know its value when we construct the AFG; determining the set of possible values is the basic question in pointer analysis. In particular, we ask which assignments are visible to a particular fetch since the only values that should be read by a fetch are those that were written earlier, barring reads of uninitialized variables.

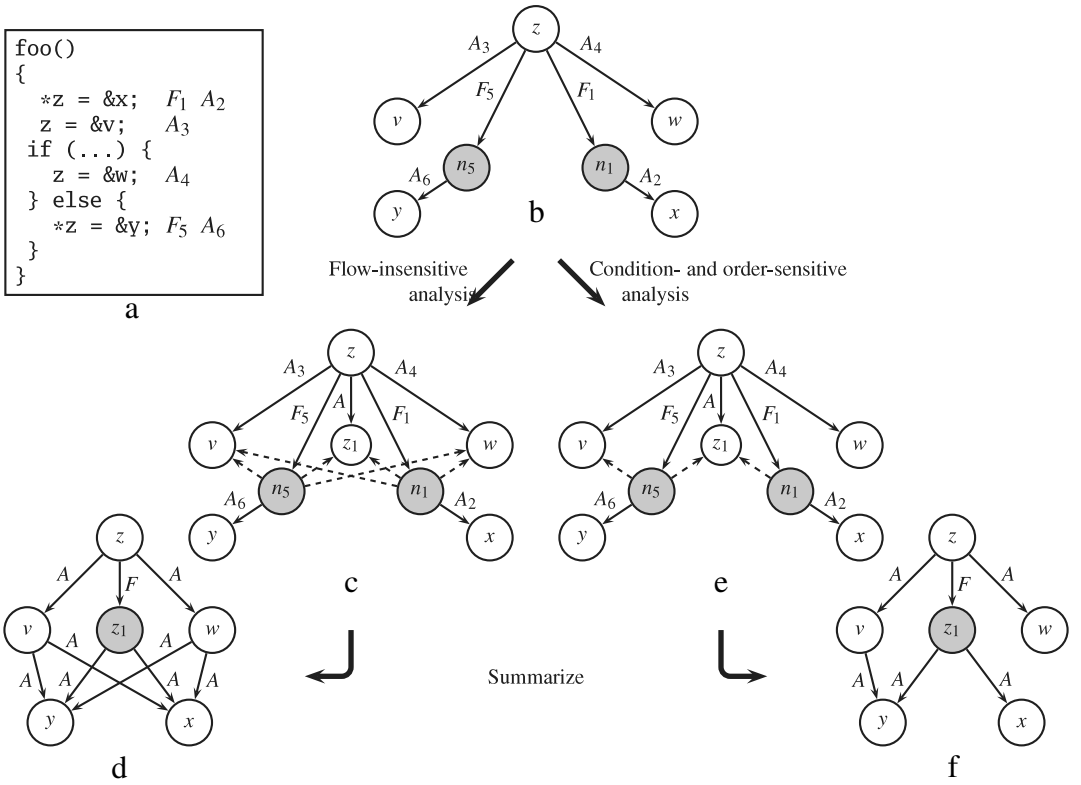


Fig. 5. An illustration of pointer-analysis using AFGs. A procedure (a) is first abstracted as an assign-fetch graph (b), whose nodes represent addresses and values and whose edges represent memory operations. Flow-insensitive analysis (c) is an abstraction where potential aliases are calculated ignoring statement order. This produces a summary (d). Considering execution order and mutually exclusive operations (e) produces a more accurate summary (f).

Answering this question is the goal of resolution, which adds alias edges from fetch nodes to location nodes to indicate what values could be fetched. Each fetch of the same variable in a procedure generates a distinct fetch node, allowing the AFG to represent variables that take on different values at different times.

Adding aliases from fetch to location nodes produces a resolved AFG. Using different analysis variations generate different resolved AFGs, such as Fig. 5(c) and (e). The former is a traditional flow-insensitive view of the program, where a fetch from a location matches any assignment to the same location; the latter is a more precise result obtained by considering statement ordering and mutually exclusive operations, which leads to a smaller number of fetch/assign matches. For example, because they appear in separate branches of a conditional, `*z=&y` and `z=&w` are mutually exclusive, so fetch F_5 cannot see assign A_4 and there is no alias edge from n_5 to w in Fig. 5(e). Also, the first fetch of z in the code (F_1) can only see the (unknown) initial value of z coming from the environment (represented by initial value node z_1). Thus, n_1 to z_1 is the only alias edge.

After resolving the AFG, we summarize it to enable its use in interprocedural analysis, which we discuss in Section 8. Broadly, we remove nodes that correspond to internal operations that are invisible to callers, such as the fetches of z – n_1 and n_5 – in Fig. 5(d) and (f), and remember only their effects. For example, in Fig. 5(e), fetch F_1 always returns z_1 (the initial value of z), represented by node n_1. We remove the (implicit) assignment of z to z_1 by the environment (the A arc in Fig. 5(e)) and replace it with a fetch edge that indicates that this fetch can only return z_1.

4.1. Pointer alias analysis

In alias analysis, executing statement `p=&r` creates the alias relation $\langle$`*p, r`$\rangle$, meaning `*p` is *an alias for* `r`. Computing points-to sets using the AFG amounts to determining the locations for which a fetch node could be an alias. We represent such relations by adding directed alias edges to the AFG; each alias edge corresponds to a potential alias relation. In Fig. 3, the dashed edge $n \rightarrow y$ indicates the alias relation $\langle$`*x, y`$\rangle$. Each alias edge starts at a fetch node and terminates at a location node.

The central goal of pointer analysis is to determine a small set of alias edges that includes every one that is actually possible (i.e., remains sound). While the most conservative over-approximation is to proclaim that every fetch node aliases every location, this would not be very helpful. Instead, the goal is to include as few relations as possible while remaining sound and computationally reasonable. In the following sections, we show how using different analysis variations within our space produces different minimal sets of alias edges for a given AFG. Each such set corresponds to a different points-to solution, with distinct precision levels.

Determining aliases between fetch nodes and location nodes is the main step in pointer analysis on AFGs. Below, we discuss a general rule of inference that we will later specialize to algorithms with varying levels of precision; that the AFG lends itself to such variants is one of its key strengths. A second strength is its ability to produce a single summary for any given procedure that can be used in any calling context.

A fetch node n can be an alias for many locations. Because a fetch can only return a value that the program wrote to memory, any alias of a fetch node must be the target of an assign edge (we model the initialization of global variables with assign edges).

We write affects(σ_A, σ_F) to indicate that the assign edge σ_A could write a value that fetch edge σ_F could read. This relation can be many-to-many: one assignment could be seen by many fetches, and a fetch might see many assignments. As with any other useful property about computer programs, *affects* is not effectively computable, so any pointer analysis must approximate it. A sound analysis demands an over-approximation: it should be true when *affects* is true, but not necessarily vice-versa. Different approximations result in different sets of alias edges for the same initial AFG, leading to different pointer analysis solutions.

When an assignment affects a fetch, the fetch can return anything written by the assignment, so aliases for the fetch must include all aliases of the assignment's "right-hand side". Formally,

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \quad \sigma_F : \alpha \xrightarrow{F} n \quad \text{affects}(\sigma_A, \sigma_F)}{\text{al}(\beta) \subseteq \text{al}(n)} \tag{ALIAS}$$

where $\gamma \xrightarrow{A} \beta$ denotes an assign edge from γ to β and $\alpha \xrightarrow{F} n$ denotes a fetch edge from α to n. The solution to pointer analysis is the minimal set of alias edges that satisfies this rule.

In Fig. 2, the assignment `p=&r` affects the fetch of `p` in statement `*p=&w`, provided that condition `c` is neglected (e.g., Fig. 2(b)), but the assignment does not affect the fetch if we consider the two statements to be mutually exclusive.

Different points in our analysis space correspond to different approximations of the *affects* relation. Below, we present an approximation corresponding to the origin of the analysis space; later sections describe different points.

5. Flow-insensitive analysis with the AFG

The [ALIAS] rule we presented above gives a precise characterization of pointer analysis that depends on the non-computable *affects* relation. Here, we describe one approximation to *affects* that corresponds to the origin of our analysis space: an Andersen-style flow-insensitive analysis.

Define the predicate *aliases* as

$$\text{aliases}(\alpha, \gamma) \quad \Leftrightarrow \quad \alpha = \gamma \ \vee \ \text{al}(\alpha) \cap \text{al}(\gamma) \neq \emptyset.$$

This says nodes α and γ are aliases if they are identical or if they are aliases for some common location node.

The *aliases* relation is a flow-insensitive (over-) approximation of the exact *affects* relation. [ALIAS] is approximated by

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \quad \sigma_F : \alpha \xrightarrow{F} n \quad \text{aliases}(\alpha, \gamma)}{\text{al}(\beta) \subseteq \text{al}(n)} \tag{FI-ALIAS}$$

Because this rule is recursive (the premise refers to the *aliases* relation, which depends on al), finding the minimal resolved AFG requires computing a fixed point. Our implementation uses the usual worklist algorithm that iterates to convergence.

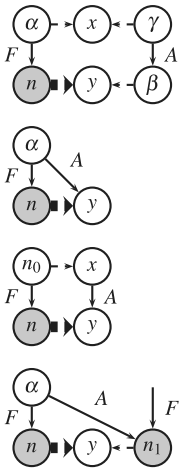
These rules correspond to an abstraction at the origin of our space: they ignore statement order and conditionals. Consider Fig. 5(c), which shows the resolved AFG for Fig. 5(b) under the [FI-ALIAS] rule. Every assign to a location is seen by all fetches from that location, so in Fig. 5(b), both n_1 and n_5 will resolve to both v and w. In the implementation, we also create the (unknown) initial value node z_1 since global z is dereferenced within the function (we lazily initialize environment variables). Nodes n_1 and n_5 also resolve to z_1.

Fig. 6 illustrates the [FI-ALIAS] rule graphically. Existing alias relationships are shown with thin dashed lines and the rule generates the edges in bold.

6. The statement order dimension

The [FI-ALIAS] rule ignores the fact that assignments and fetches in a program happen in sequence: later assignments cannot be seen by earlier fetches. This leads to overly approximate results, arriving at more aliases than actually possible.

To increase precision this section considers the ordering of statements. In general, this is not a partial order, but becomes one in the case of loop-free procedures. For this reason, before pointer analysis, we convert all loops into tail-recursive procedures (we describe this in detail in Section 9). This conversion benefits all algorithms because problems that may be unsolvable in general become solvable for loop-free programs; or problems that require nonlinear algorithms in general can



The most general case: β is assigned to γ, n is fetched from α, and both α and γ may hold the address of x; therefore y is added to $\mathrm{al}(n)$, since $y \in \mathrm{al}(\beta)$.

The simplest case: y is assigned to α and later fetched as n (here, $\beta = y$ and $\alpha = \gamma$), so y is added to $\mathrm{al}(n)$.

When y is assigned to x, and later n is fetched from n_0, which may contain the address of x, y is added to $\mathrm{al}(n)$. This is when $\alpha = n_0$, $\gamma = x$, and $\beta = y$.

When a value n_1, which may be the address of y, is assigned to α, and n is the result of fetching α, then n may also take the value y. Here, $\alpha = \gamma$ and $\beta = n_1$.

Fig. 6. Visualizing the [FI-ALIAS] rule. The bold alias edges (dashed lines) are added if the rest of the pattern exists.

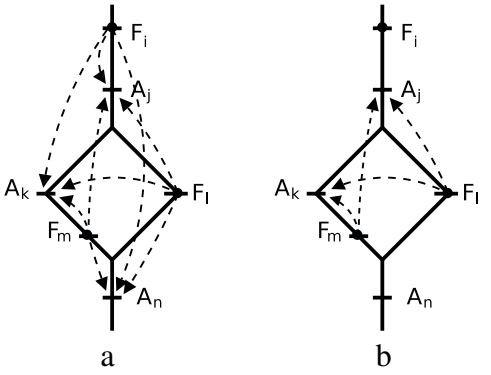


Fig. 7. Ordering information reduces the number of alias edges.

be solved by a linear algorithm for acyclic graphs. By separating out loops, the loop-free portions can benefit from the more efficient algorithms, while only loops suffer the less efficient, or inaccurate ones.

For the rest of the paper, we assume a given partial order of all the statements in a procedure. The partial order arises directly from the control flow graph of a procedure: for two statements σ_1 and σ_2, $\sigma_1 \sqsubseteq \sigma_2$ if σ_2 is reachable from σ_1. This section refines the approximation of the *affects* relation by considering this partial order. An assignment σ_A cannot affect a fetch σ_F if $\sigma_F \sqsubseteq \sigma_A$.

Fig. 7 illustrates the matching of fetch and assign operations when considering the statements partial order. This figure presents two identical fragments of the control-flow graph of a program showing some assignments and fetches. For simplicity, assume that these operations are performed on the same program variable, so that *aliases* is always true. For the sake of presentation *only*, dashed arrows have been added to the control-flow graph—a dashed arrow from a fetch F_i to an assign A_j indicates that F_i resolves to A_j.

Fig. 7(a) correspond to the matchings when the [FI-ALIAS] rule is applied, whereas the dashed arrows in Fig. 7(b) indicate the matchings obtained when execution ordering is considered. For instance, note that fetch F_i will only resolve to assignments (omitted) occurring before the code fragment shown, and assignment A_n does not affect any of the fetches depicted since it happens later.

The refined affects relation which "unfolds" the partial order axis of our analysis space is defined as

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad \mathrm{aliases}(\alpha, \gamma) \qquad \sigma_A \not\sqsupseteq \sigma_F}{\mathrm{al}(\beta) \subseteq \mathrm{al}(n)} \qquad \text{(PO-ALIAS)}$$

Informally, in addition to α and γ being aliases, the [PO-ALIAS] rule also requires that the assign σ_A does not occur after the fetch σ_F in the partial order. The idea of unfolding the horizontal axis is that different approximations to the execution order can be given: from a "flat" set of statements where no statement precedes another, to the full partial order obtained from the program text. Actually, we found that there is no practical advantage in considering anything less than the full partial order. That extreme point yields both more efficient and more precise results that any other point on the statement-ordering axis.

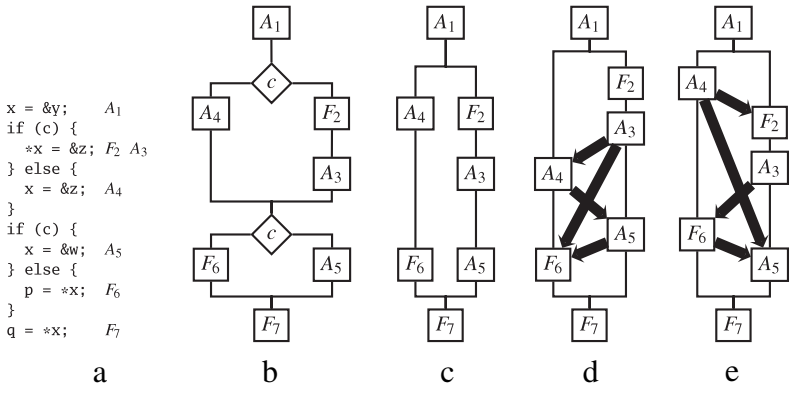


Fig. 8. Motivation for flow-aware analysis. (a) A (nonsensical) program with two correlated conditional branches. (b) Its control-flow graph. (c) Considering correlation of conditions. (d) One possible total ordering under flow-aware ordering (subscripts indicate the order). Bold arrows show spurious dependences. (e) Another possible ordering ($A_1, A_4, F_2, A_3, F_6, A_5, F_7$).

Fig. 7(b) shows the matchings obtained when applying the [PO-ALIAS] inference rule. For example, F_m matches A_k and A_j because both assignments precede F_m; F_i does not match any of A_j, A_k, and A_n because all of these assignments happen after F_i.

Note that the [PO-ALIAS] rule does not require the fetch to occur *after* the assignment, or $\sigma_A \sqsubseteq \sigma_F$. Thus, an assignment is considered to affect a fetch occurring on a different branch. This is the case with F_l and A_k in Fig. 7(b). In order to obtain an answer from the "$\sqsubseteq$" relation, [PO-ALIAS] requires instead that $\sigma_A \not\supseteq \sigma_F$, which is true in case σ_A and σ_F are not comparable. This is the reason: When conditions are ignored, two branches of an *if* statement are treated the same as two branches of a parallel construct; two statements on different branches of a parallel construct can indeed affect each other.

Also note that the [FI-ALIAS] rule is an (over)approximation of the [PO-ALIAS] rule where "$\sigma_A \not\supseteq \sigma_F$" is proclaimed true for any given pair σ_A and σ_F. Intuitively, this means that the fixed-point obtained by [PO-ALIAS] is smaller than that of [FI-ALIAS]. Given an initial AFG, the iterative process of adding alias edges until convergence terminates earlier for [PO-ALIAS], which means a more precise, and in some cases *more efficient*, analysis.

6.1. Flow-aware analysis

A central advantage of our AFG is that it makes it easy to implement different abstractions. In this section, we present a particularly interesting one: flow-aware analysis [15], which supplements statement ordering with a total order obtained by topologically sorting a procedure. This is a sound approximation in the common case when all branching is due to conditionals, not parallelism. The fact that two branches of a conditional statement are mutually exclusive can be expressed by adding extra $\sqsubseteq$ relations on top of those obtained from the control flow of the given procedure. Pointer analysis algorithms obtained in this way are between points (d) and (e) of Fig. 2. The more such extra $\sqsubseteq$ relations are added, the higher along the conditions axis is the resulting analysis. That makes it more precise, and interestingly also more efficient. This efficiency is due to the simplicity of the total order—the affects relation is approximated by a simple integer comparison; in contrast, taking into account program conditions in full generality requires many more computational resources.

Fig. 8 illustrates this on a program fragment. The original partial order is obtained from the flow graph of Fig. 8(b). For example, $A_3 \sqsubseteq F_6$, although in real execution F_6 can never follow A_3 because the two *if* statements have the same condition c. Fig. 8(c) shows how Fig. 8(b) could be rewritten if the correlation between the two *if* statements were taken into account. The approximation presented in this section is more precise than Fig. 8(b) in that it allows us to express that A_4 does not affect F_2, but it does not have the precision of Fig. 8(c) in that it still allows the possibility that A_3 affects F_6.

With flow-aware analysis, the statements might be arranged in the total order of Fig. 8(d), which captures the fact that A_4 does not affect F_2. It does so by (arbitrarily) placing A_4 after F_2 and A_3 in the total order. This amounts to adding the arc $A_3 \sqsubseteq A_4$ highlighted in Fig. 8(d). This total order adds other $\sqsubseteq$ relations, also highlighted in Fig. 8(d). The choice of ordering between conditional branches is non-deterministic; Fig. 8(e) is another possible order.

Approximating statement ordering with a total order has the benefit of disassociating one branch of a conditional from the other (but not vice-versa). For example, in Fig. 8(d), the total order correctly models the fact that A_4 is not visible to A_3, but makes A_3 visible to A_4. The total order in Fig. 8(e) has the opposite problem. The first author's thesis [14] shows how a pair of orderings can capture exclusivity of conditional branches for practically all programs. More precisely, it is exact for programs with planar control-flow graphs and merely a safe approximation for all others. We do not bother to test programs for planarity since we use the same ordering scheme for all programs.

Formally, let $\text{affects}_0(\sigma_A, \sigma_F)$ be true when assignment σ_A occurs before fetch σ_F in the total order. The [ALIAS] rule for flow-aware analysis is

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad \text{aliases}(\alpha, \gamma) \qquad \text{affects}_0(\sigma_A, \sigma_F)}{\text{al}(\beta) \subseteq \text{al}(n)} \quad \text{(FLOW-AWARE)}$$

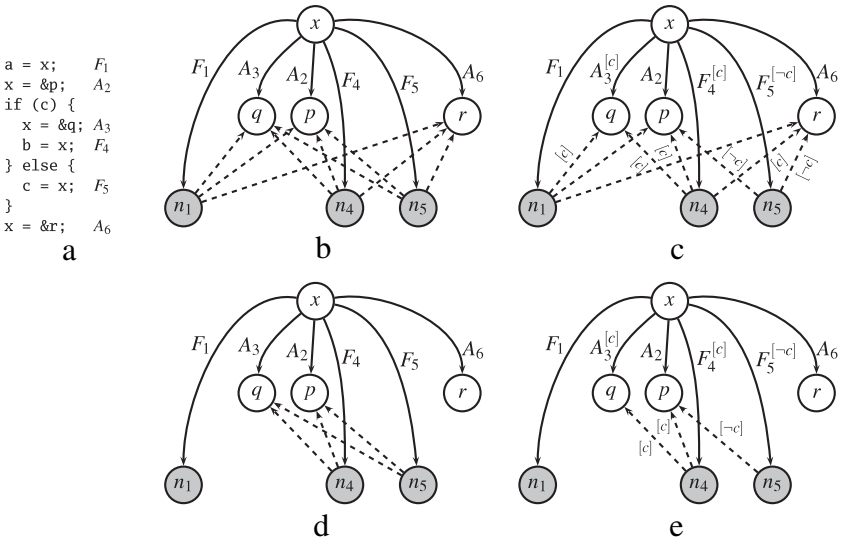


Fig. 9. The effect of considering ordering and predicates. A program fragment (a) and its resolved AFG under flow-insensitive analysis (b) without and (c) with predicates, flow-aware (d) without predicates, and (e) with predicates.

This rule leads to faster convergence with fewer alias edges when compared to [FI-ALIAS] because it prunes off many spurious dependencies that would otherwise have to be considered.

To implement such analysis, we simply augment each edge σ in the AFG with an index $\text{rank}(\sigma)$ from a topological sort of the statements in the procedure (each procedure is acyclic after our preprocessing). Then $\text{affects}_0(\sigma_A, \sigma_F)$ can be implemented as $\text{rank}(\sigma_A) < \text{rank}(\sigma_F)$.

7. The condition dimension: Guards on arcs

Here, we describe a technique for more precisely treating the behavior of conditionals in a program. We assume guards on fetch and assign edges to indicate under what conditions they may run. This contributes to the precision of the analysis by disallowing matches between fetches and assigns with mutually exclusive guards. The job of pointer analysis is then to calculate not only whether an alias can occur, but also under what conditions.

Fig. 9 illustrates the advantages of considering conditionals. Even with a flow-insensitive abstraction, modeling conditionals can eliminate spurious aliases: Fig. 9(b) is the resolved AFG under the flow-insensitive abstraction, the point at Fig. 2(b). In Fig. 9(c), we added predicates to the fetch and assign edges in conditional branches, then used them to eliminate spurious aliases. This is the point of Fig. 2(f). For example, the alias arc from n_5 to q in Fig. 9(b) is absent from Fig. 9(c) since it is impossible: A_3 only runs when c is true and F_5 only runs when c is false (they are in opposite branches of the *if*).

Considering the effect of conditionals in addition to considering statement ordering provides an additional improvement. For example, Fig. 9(d) is the resolved AFG under the flow-aware abstraction. This allows us to prune the aliases for n_1 since F_1 appears first, and for r since A_6 appears last. Further considering conditionals gives Fig. 9(e), in which the alias from n_5 to q has been removed because F_5 and A_3 cannot run simultaneously (they are in different branches of the *if*).

Formally, we allow every edge in an AFG to have a condition; we write $\mathcal{C}(\alpha, \beta)$ to denote the condition on an edge from α to β. Conditions on assign and fetch edges are assumed given (extracted from the program). Pointer analysis extends them to conditions on alias edges.

To this point, we considered an alias edge between a pair of nodes either to exist or not to exist. Now, for simplicity of notation, we assume an alias edge between every pair of nodes; some alias edges may have the always false condition, which is a representation of non-existence. A pointer analysis ignoring conditions is a special case of one considering conditions, namely any condition that is not always false is approximated to be always true.

Analysis considering conditions starts by assigning the false condition on the alias edge between every pair of nodes; the only exceptions are the self-loops on location nodes—they have the condition always true. Then, the conditions on all the alias edges are calculated as the least fixed point of the following rule

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad \sigma_A \not\sqsupseteq \sigma_F}{\mathcal{C}(\alpha, n) \wedge \mathcal{C}(\alpha, x) \wedge \mathcal{C}(\gamma, x) \wedge \mathcal{C}(\gamma, \beta) \wedge \mathcal{C}(\beta, y) \subseteq \mathcal{C}(n, y)} \quad \text{(GUARDED-ALIAS)}$$

This rule can be best understood by considering the first and most general case of Fig. 6. The premise of the rule assumes an assign and fetch edge satisfying the ordering constraint of Section 6. The conclusion of the rule states that for any two

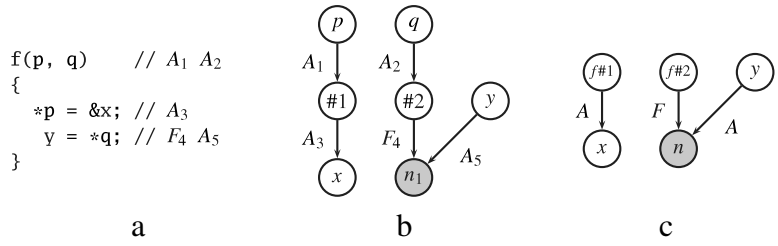


Fig. 10. Handling arguments: (a) a function, (b) its AFG, and (c) its summary.

nodes x and y the AND of the conditions on all the five existing edges in Fig. 6 must be included in the condition of the new alias edge from n to y.

Please note that when conditions are ignored, i.e., every alias edge has the condition always true or always false, then [GUARDED-ALIAS] reduces to the rule [PO-ALIAS].

As with the program's partial order, different approximations can be given to program predicates as well as to the results of the above Boolean operations. A widening operator [44] can be defined such that, e.g., the logical *or* of two predicates is widened to true. This is one of the mechanisms we use in our implementation, controlled by a parameter.

In our experimentation, the most general treatment of conditions proved prohibitively expensive. The experiments reported in Section 10 were run with the following approximation. Any condition on an alias edge, unless always false, was replaced by the always true condition. In contrast, conditions on assign and fetch edges were kept accurate, as obtained from the source program. In addition, conditions were not propagated interprocedurally.

With this treatment of the conditions, rule [GUARDED-ALIAS] becomes

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad \text{aliases}(\alpha, \gamma) \qquad \sigma_A \not\supseteq \sigma_F \qquad \mathcal{C}(\gamma, \beta) \wedge \mathcal{C}(\alpha, n)}{\text{al}(\beta) \subseteq \text{al}(n)}$$

(SIMPLIFIED-GUARDED-ALIAS)

That is the same rule as [PO-ALIAS] with the addition of the requirement $\mathcal{C}(\gamma, \beta) \wedge \mathcal{C}(\alpha, n)$. Such Boolean reasoning was performed by the BEAM theorem prover [45].

8. Interprocedural analysis

So far we have only described how to analyze single procedures. Here, we describe how to extend our analysis to work across procedures. We explain how to summarize procedures, handle arguments, and how to use summaries at call sites. Our summaries do not assume anything about whether arguments are aliased and as such can be used in any setting and remain sound, a key advantage of our approach.

8.1. Computing summaries

After computing aliases, we prepare a procedure's AFG to be used at a call site by summarizing it. We delete anything that a caller could not see, such as temporary memory fetches n_1 and n_5 in Fig. 5(c). Before we delete such nodes, we transfer their effects to nodes that will remain. Fig. 5(d) shows this. In general, if an assignment is made to a fetch node n, and n can be an alias for a location node n', the assignment is equivalent to one to n'. For example, in Fig. 5(c), n_1 is assigned the address of `x` and can be an alias for z_1, `v`, and `w`, so we add assign edges from z_1, `v`, and `w` to `x`. Similarly, we add edges from z_1, `v`, and `w` to `y`. Finally, we remove n_1 and n_5 and "demote" z_1 to a fetch node to indicate the dereference of `z`. This produces the flow-insensitive summary in Fig. 5(d).

8.2. Modeling arguments

Argument passing in the AFG is represented the same way as in the given flow graph, which is somewhat different from what a compiler does. During actual execution a caller would place argument values on the stack, and then invoke the callee. The callee would then access those argument values by fetching the contents of its parameters. In our representation, the callee first allocates its parameters like any other local variables, and then explicitly assigns the argument values to those parameter variables. Thus, the argument values are the initial values of the parameter variables. This is analogous to the treatment of global variables; for them we also create an assignment of new values representing the initial values at entry to the callee.

Fig. 10 illustrates how we handle arguments. We add location nodes for formal arguments p and q and initial value nodes #1 and #2 that represent their initial (caller-passed) values. Fig. 10(b) also includes nodes and edges for the two statements.

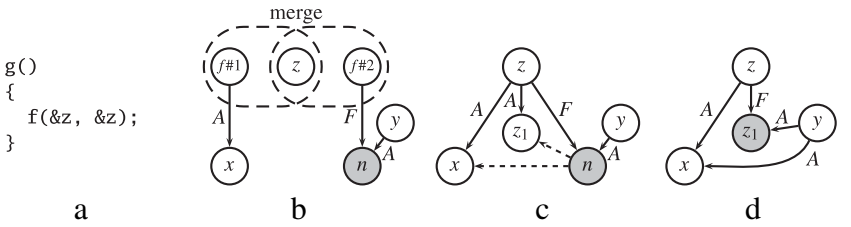


Fig. 11. Handling procedure calls: (a) a procedure g, which calls f (Fig. 10), (b) its initial AFG, (c) after resolving, and (d) after summarizing.

Since formal arguments are initialized by "#i" nodes, the AFG representation for `*p = &x` in Fig. 10(b) does not include a fetch edge for p; `*p` directly yields its initial value, #1.

In summarizing this (alias-free) procedure, we remove the nodes for the formal arguments p and q and rename the initial value nodes to include the procedure's name. Also, fetch edge F_4 in Fig. 10(b) generates an initial value for node #2, which in the final summary Fig. 10(c) we label n.

8.3. Modeling procedure calls

When building the initial AFG for a procedure, we replace a call to a function with the summary for the function. Instantiating a summary involves merging any global variables shared by both caller and callee and connecting formal to actual arguments. Fig. 11 illustrates calling procedure f (Fig. 10). The address of global variable z is passed as both formal arguments p and q, so when we copy the summary of f from Fig. 10(c), we mark the nodes for the initial values of p and q, $f\#1$ and $f\#2$, to be merged with z.

We perform the same process for each global variable: its node in the callee is merged with its node in the caller. This is vacuous in Fig. 11 since g does not touch globals x or y.

Node merging works even when an actual argument is an expression. The call $f(z, z)$ would represent its actual arguments as two fetch edges from z to nodes, say, n_1 and n_2, which would be merged with the value nodes for the formals: $f\#1$ and $f\#2$. Computing aliases on the AFG would find the two arguments aliased.

After instantiating each callee's summary, we compute the caller's summary. In Fig. 11(c), we added an initial value node for global z and used flow-insensitive analysis to add alias edges from n to x and z_1. We produced the summary in Fig. 11(c) by removing fetch node n; its aliases now manifest themselves as the assign edges from y. A caller of g can know from this summary that g dereferences z.

This example illustrates how a summary is agnostic about argument aliasing and can be used in any context. The summary in Fig. 10(c) treated arguments p and q as distinct, but we merged them at the f call site in g and found that running g makes y point to x. Other published solutions to the environment problem either use information from the environment while building a summary [40], or build multiple summaries for each function, one for each possible environment [42,43].

8.4. Interprocedural flow-aware ordering

This section illustrates how we can manipulate ordering information across procedures and how spurious pointer relations that would otherwise span multiple functions are avoided. We will focus on flow-aware analysis.

Performing flow-aware analysis across procedure calls requires us to order statements on both sides of a call site. To get this right, we shift the indexes of the callee by the maximum index that occurs in the caller before the call site, then increase the indexes in the caller that appear after the call (based on the maximum index within the callee's summary). This means that an assignment occurring after the function call cannot be seen by a fetch occurring before or within the called function. Alternatively, a fetch occurring before the call site does not read a value assigned later by the callee. Fig. 12 illustrates this.

In Fig. 12(b), `x`'s value is read by `q=x` in `bar()` then modified by `f()` at the call site `f(&x)`. When statement `*q=&w` executes, the original value of `x`, `&v`, is set to point to `w`. By ignoring order information, an interprocedural flow-insensitive analysis would pessimistically include `a` and `b` as values that could be read by `q=x`. Our flow-aware analysis avoids such spurious relations. Fig. 12(c) and (d) show the resolved and summary AFGs for function `f`. Nodes labeled "#1" in Fig. 12(c) and "`f#1`" in Fig. 12(d) represent the initial value for argument `p`, which is merged with the argument node at the call site (modeling arguments and summary instantiation is explained in detail in Section 8.2).

We sort the edges in a function summary and number them starting from 1, being careful to preserve the relative order among statements. Fig. 12(e) shows how the summary for `f` is instantiated at the call site `f(&x)`. Statements before the call are labeled A_1, F_2, and A_3.

To place a callee's statements in the total order, we add the highest index before the call to every statement in the callee's summary when we instantiate it. In Fig. 12(e), this index is 3, so we label callee's statements $A_{1\backslash 4}$ and $A_{2\backslash 5}$ to indicate that A_1 and A_2 will become A_4 and A_5. Processing the remaining statements after the function call gives the initial AFG for `bar()` in Fig. 12(f).

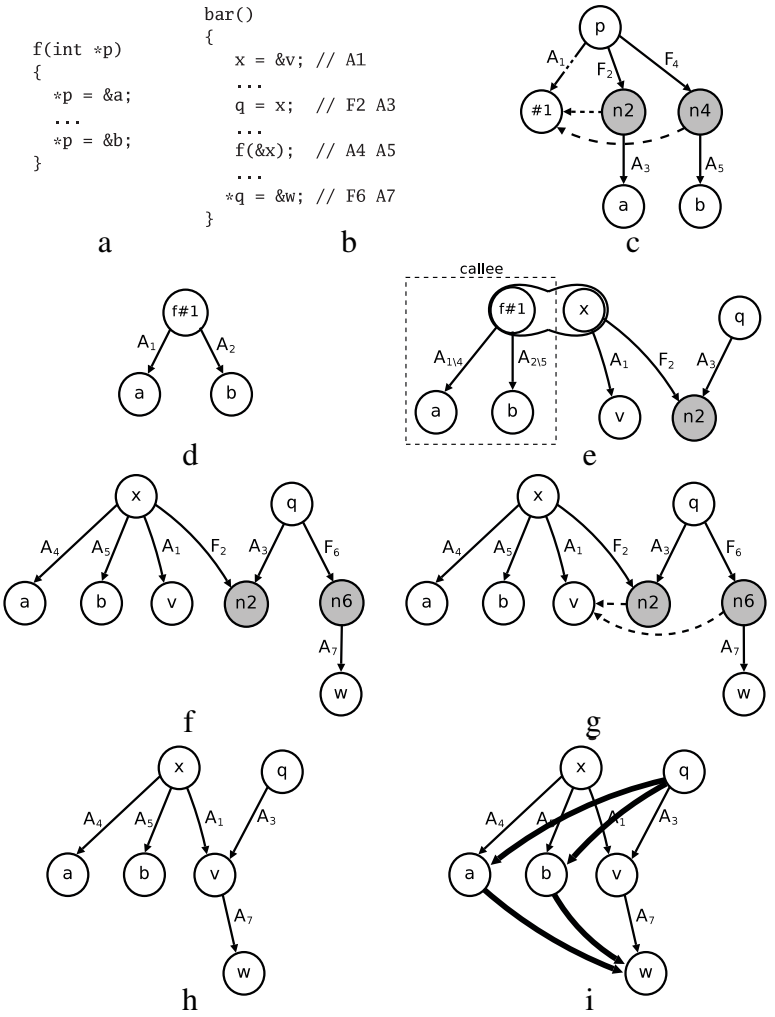


Fig. 12. Propagating flow-aware ordering across procedure calls. (a) Function `f()` is called by (b) `bar()`. The resolved (c) and summary (d) AFG for `f()`, which is instantiated at the call site `f(&x)` (e): the indexes in the summary are updated and the remaining statements in `bar()` are processed, generating the initial AFG for `bar()` (f). Flow-aware analysis is performed (g) by considering ordering across procedures. The flow-aware points-to set (h) is more precise than the flow-insensitive (i).

Fig. 12(g) shows the result after flow-aware analysis. Note the fetch of `x` in `q=x` (F_2) resolves to A_1, the only assignment occurring before that fetch. When F_6 matches A_3 a single alias edge is added. Fig. 12(h) shows the corresponding points-to sets, which is more precise than the flow-insensitive result in Fig. 12(i).

9. Loops and recursive procedures

This section describes how loops and recursive procedures are handled in general, emphasizing the case for flow-aware analysis. We convert loops into tail-recursive procedures and iteratively analyze (such) recursive procedures until we reach a fixed-point. The first time a recursive procedure is analyzed, we do not have a summary for it, so we only consider the other statements in the procedure. This gives a better summary for the procedure, which we then instantiate at recursive call sites and summarize again.

It may appear that this process might not terminate, but this is not the case. It turns out that the number of edges and arcs that can be added is bounded. The number of heap nodes is bounded because of the heap naming scheme we adopt [14]. The number of fetch edges is bounded because the final summary allows at most one fetch edge out of any node, and there is a user-settable limit on the length of any chain of fetch edges. Finally, we prohibit duplicate assign edges. Together, these constraints bound the summary and guarantee convergence. If duplicate assignments between a pair of nodes is allowed, such as in Fig. 13(e), the comparison between two summaries must only consider whether $x \xrightarrow{A} y$ exists and not the number of such edges.

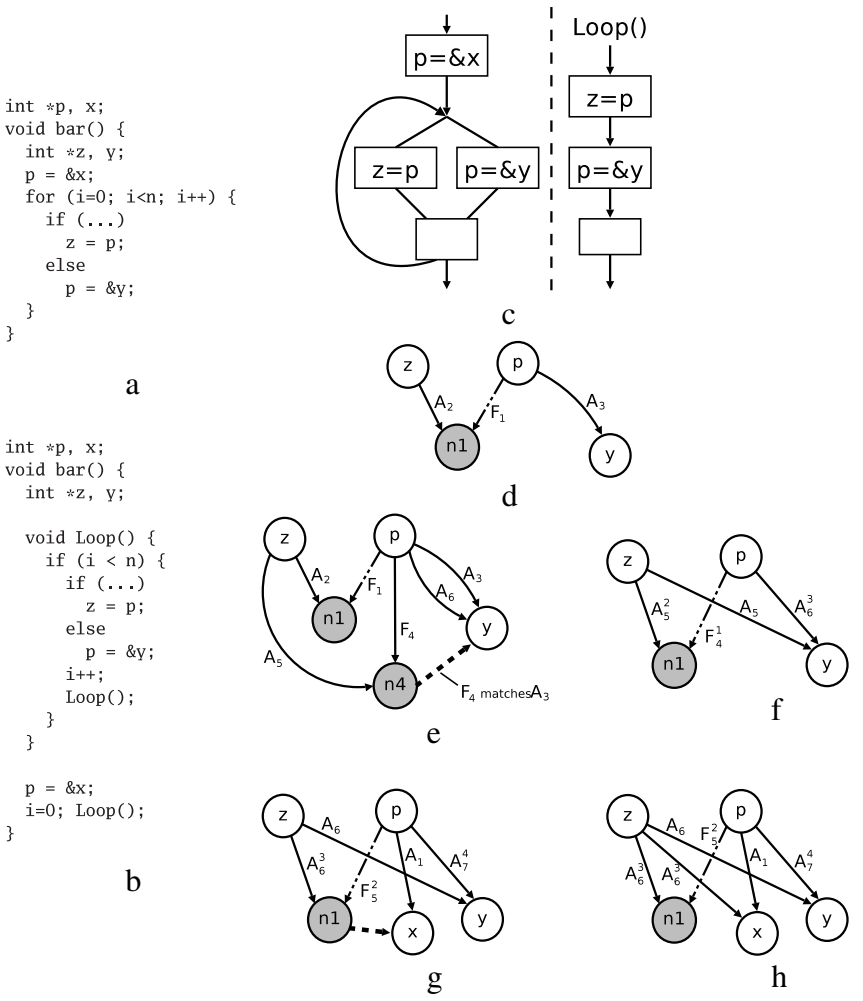


Fig. 13. Handling loops and recursive functions.

Fig. 13 illustrates summarizing a simple *for* loop. We transform the function in Fig. 13(a) into the tail-recursive procedure in Fig. 13(b). The transformation assumes an Algol/Pascal-style scope where a procedure can be defined inside another procedure, and variables declared in between are local to the outer procedure and global to the inner one. To illustrate, we nested the definition of `Loop` inside `bar` to emphasize that it has access to `bar`'s local variables.

Fig. 13(c) is a simplified control-flow graph for this code. On the left is the structure of the loop; on the right is a linearized version of the `Loop` procedure that assumes flow-aware analysis placing the *then* branch of the *if* before the *else*.

Fig. 13(d) is the first summary of `Loop`—the assignment `p=&y` is hidden from `z`. We now have a summary of `Loop`, which is used in the second iteration of the analysis giving Fig. 13(e). Edges with subscripts 1, 2, and 3 correspond to the loop body statements within the function. Instantiating the earlier summary adds edges F_4, A_5, and A_6 (the indexes are shifted as in Section 8.4). This time, fetch edge F_4 matches assignment A_3, and `z` will point to `y` as a result; edges F_4 and A_3 belong to different iterations of the original loop.

Fig. 13(f) is the summary of Fig. 13(e) and also the fixed-point—the final summary for function `Loop()`. Some edges have two numerical labels because they are the result of merging multiple edges. This means they represent the interval for which the operation is valid. For example, A_6^3 represents the merge of A_3 and A_6.

Fig. 13(g) shows the graph for `bar` after we inserted the summary for `Loop`. The fetch of `p` resolves to `p=&x` since the assignment occurs before the fetch (i.e., we check that the subscript index of the fetch is greater than the superscript on the assign, in case one exists, or the subscript otherwise). Finally, Fig. 13(h) shows the summary for `bar`, which notes that the global variable `p` is fetched.

Note in Fig. 13(g) that F_5^2 does not resolve to A_7^4, although $5 > 4$. This happens because A_1 is the only *new* fact that needs to be considered when instantiating the summary of `Loop()` into `bar()`; all the other edges were already present in Fig. 13(f), and F_4 had already been resolved to A_3 in Fig. 13(e). Our algorithm is such that only the addition of new aliases or new facts trigger the inference rule for resolution. Details can be found in the first author's thesis [14].

Table 1
Benchmarks: whole applications and Linux 2.6.23 kernel modules.

Name	Version	Description	Lines	Functions	
				Source	Internal
balsa	2.3.13	E-mail client	110.0k	2659	4682
bftpd	2.0.3	FTP server daemon	4.9	145	245
bison	2.1	GNU parser generator	25.4	700	1297
black-hole	1.0.9	Spam prevention	18.0	87	290
cfingerd	1.4.3	Configurable "finger" daemon	4.5	68	123
compress	1.3	Compression software	2.2	30	66
firestorm	0.5.4	Network firewall	8.0	229	330
gzip	1.2.4	File compressor	8.3	126	331
identd	1.3	TCP identification protocol	0.3	21	40
ispell	3.1	Spell checker	10.1	117	337
lhttpd	0.1	Http server and content management	0.8	20	40
make	3.81	Application building system	22.1	309	853
mingetty	1.07	Minimalist "getty" program	0.4	24	43
muh	2.05d	IRC bouncing tool	5.2	75	107
pcre	7.1	Regular expression interpreter	15.4	63	300
pgp4pine	1.76	PGP for the Pine mail reader	4.2	72	146
polymorph	0.4.0	Filename converter ("unixizer")	1.0	19	31
stunnel	3.26	Universal SSL tunnel	3.9	93	134
tar	1.15.1	File archiver	32.7	651	1145
trollftpd	1.26	FTP server daemon	2.8	52	102
algos		I^2C module	1.7k	52	75
amso1100		Network module	8.2	234	304
atm		I^2C module	5.7	166	209
bluetooth		Bluetooth module	8.4	291	358
busses		I^2C module	20.3	434	520
chips		I^2C module	7.1	115	131
core		USB module	15.8	527	659
cxgb3		Network module	9.3	307	420
gadget		USB module	41.9	693	872
host		USB module	25.6	629	878
ieee1394		Firewire module	24.0	512	735
inficore		Network module	25.4	659	840
irq		Kernel module	2.3	86	105
kernel		Kernel module	64.5	2451	3254
misc		USB module	16.8	523	665
mlx4		Network module	4.3	149	178
mthca		Network module	15.0	442	597
power		Kernel module	4.8	220	302
serial		USB module	42.1	911	1205
storage		USB module	12.2	274	366
video		Video module	84.0	1134	1667

10. Experimental results

We implemented our pointer analysis framework in a static analysis (bug-finding) tool called BEAM [46] developed at IBM. In this section, we show two types of experimental data: analysis metrics and bugs we found when analyzing several benchmark applications.

The analysis metrics are also used to validate our hypothesis that flow-insensitive analysis is not the most efficient algorithm for pointer analysis. Indeed, our abstraction model and AFG uncovers analysis variations that are more efficient and more precise than (Andersen-style) flow-insensitive analysis.

Some of the applications we tested receive regular manual and automated source code checking, so finding errors in a number of them reinforces the utility of our analysis.

Table 1 lists the benchmarks we tested. There are two kinds: complete applications (executables) and Linux kernel modules. The *lines* column lists the number of source lines of code we analyzed. The *source functions* column lists the number of functions defined in the source; *internal functions* refers to the number of functions after loops have been transformed into tail-recursive functions. This latter number is more representative of the control complexity of the program.

10.1. Running time

Table 2 shows the times it took to perform pointer analysis using variants of our algorithms. The times in the flow-insensitive column are our baseline, which we collected by running BEAM on a 2.2 GHz Pentium 4 machine with 4 GB of memory running Linux. These times include calculating the points-to sets as initial, resolved, and summary AFGs, and propagating summaries until convergence, but do not include the time for BEAM to read source files, parse, and build the IR.

Table 2
Improvements in time and accuracy for flow-aware analysis.

Name	Flow-insensitive	Improvement for flow-aware			
		Condition-insensitive		Condition-sensitive	
	Time	Time	Accuracy	Time	Accuracy
balsa	43s	15%	10.95%	−40%	12.01%
bftpd	3.5	9.5	1.09	−22	1.09
bison	51.2	55	27.34	−88	29.24
blackhole	11.2	7	7.36	−52	7.36
cfingerd	2.6	12	1.07	−20	1.07
compress	0.7	3.6	1.05	−11	1.05
firestorm	1.3	4.3	9.42	−16	9.42
gzip	1.4	8.7	6.04	−17	7.23
identd	0.1	4.5	0.16	−15	0.16
ispell	8.0	50	10.99	−35	11.44
lhttpd	1.5	0.7	3.00	−5.6	3.00
make	120	67	11.48	−90	12.89
mingetty	0.4	0.8	0.1	−8.2	0.1
muh	2.8	25	7.31	−14	7.31
pcre	19	83	15.53	−94	15.53
pgp4pine	2.9	5.0	1.04	−26	1.04
polymorph	0.5	1.8	0.32	−23	0.32
stunnel	1.5	3.9	10.33	−18	10.33
tar	37	43	6.56	−51	6.71
trollftpd	1.8	0.13	2.94	−21	2.94
algos	0.5	6.9	1.23	−17	1.31
amso1100	1.5	8.4	8.14	−5.3	9.64
atm	1.6	0.01	3.89	−10	3.89
bluetooth	1.2	7.4	6.47	−5.4	7.28
busses	7.3	14	25.68	−8.5	25.68
chips	2.0	8.3	2.51	−8.2	2.51
core	5.6	5.8	10.78	−12	10.78
cxgb3	2.4	6.1	14.29	−22	14.29
gadget	18.9	31	70.09	−53	73.66
host	16.6	35	15.53	−33	15.53
ieee1394	34.9	8.0	27.59	−40	28.69
inficore	7.3	7.3	18.67	−5.0	18.67
irq	0.4	1.3	1.71	−17	1.71
kernel	27.5	14	13.84	−145	16.50
misc	6.9	6.6	4.20	−12	4.20
mlx4	0.9	1.9	5.01	−20	5.01
mthca	3.7	13	8.81	−9.2	8.81
power	1.5	8.4	4.33	−26	4.33
serial	12.7	33	26.69	−22	26.69
storage	4.8	16	5.07	−17	5.07
video	31.2	38	21.75	−39	23.36

In general, analysis times are short enough to make our technique practical for bug finding (although perhaps not for compilation), but vary widely depending on the size of the input program and other characteristics. An important characteristic is the shape of the program's call graph—an application with big clusters of mutually recursive functions makes propagating summaries a more challenging task since a global fixed point is necessary for the analysis to converge (i.e., all the summary AFGs for all the functions in the strongly connected component have to stabilize).

Table 2 also shows how analysis times improve or worsen for different analysis algorithms. All the speedup numbers are in percentage improvement over the baseline flow-insensitive analysis for various flow-aware algorithms. We report speedups when ignoring and considering conditionals, which are modeled as being true, false, or unknown when summaries are computed.

Heeding conditions consistently takes more time than ignoring them. This is not surprising since it requires invoking a theorem prover [14]. But using a flow-aware analysis is consistently faster than flow-insensitive analysis.

10.2. Accuracy

Table 2 also presents a measure of how the quality of results improve when different abstractions are used. Because all our abstractions are sound (i.e., guaranteed to never miss an actual dependency), the fewer assign edges our algorithms report, the better. Thus, the numbers in Table 2 report the percentage reduction in the ratio of assign edges to nodes in the summary graphs over the value for the baseline—flow-insensitive analysis. Larger numbers are better.

Unfortunately, this percentage decrease in edge ratio is only a crude measure of the quality of the results. For example, it does not consider that some analysis variations provide conditions attached to assign and fetch edges and that some

Table 3
Bugs found.

Name	Number of		Type
	Reports	Bugs	
balsa	8	2	null pointer dereference, passing null argument
bftpd	2	2	memory leak, using garbage value from malloc
bison	5	0	
black-hole	13	3	memory leak
cfingerd	7	1	memory leaks when things fails elsewhere, file leak
compress	0	0	
fireStorm	8	0	
gzip	4	0	
identd	1	0	
ispell	5	1	null pointer dereference
lhttpd	0	0	
make	14	1	null pointer dereference
mingetty	0	0	
muh	2	0	
pcre	8	2	pointer to local variable exposed, null pointer
pgp4pine	19	3	null pointer dereference
polymorph	3	0	
stunnel	6	1	file leak
tar	9	0	
trollftpd	15	3	null argument passing, double allocation
algos	0	0	
amso1100	2	1	null pointer dereference
atm	0	0	
bluetooth	2	0	
busses	2	1	null pointer dereference
chips	5	0	
core	3	1	pointer to local variable exposed
cxgb3	5	2	null pointer dereference
gadget	9	2	null pointer dereference
host	7	1	null pointer dereference
ieee1394	3	0	
inficore	5	1	null pointer dereference
irq	7	1	null pointer dereference
kernel	11	2	pointer to local variable exposed, null pointer
misc	8	2	pointer to local variable exposed, null pointer
mlx4	4	0	
mthca	6	2	null pointer dereference
power	0	0	
serial	4	1	null pointer dereference
storage	0	0	
video	9	2	buffer overrun, null pointer dereference
Total	221	38	

solutions provide ordering information. Flow-insensitive and flow-aware solutions for a given program may end up having the same number of edges, but access to order information can be a big advantage. Similarly, being able to distinguish what happens under a given condition by taking advantage of the predicates attached to the graph edges is key in certain cases.

10.3. Bug reports

More effective bug finding, specifically pointer-related bugs, is the main goal of our work. Here, we discuss the sorts of bugs we found and the analysis it took to find them.

Table 3 lists the number of and character of pointer-related bugs we found, which include null pointer dereference, returning from a function with a global variable referring to the function's local variable, memory leaks, and buffer overruns.

In all these runs, our pointer analysis was used for MOD analysis, i.e., calculating function side-effect. They were all run in the flow-aware, condition-insensitive mode. A bug gets reported only if it can be confirmed by symbolic execution, which is the most accurate form of analysis. It is independent of the pointer analysis described in this paper, and it is too expensive to be used on more than the actual function containing the bug. Our pointer analysis supplies the symbolic execution with approximate information about the side-effects of called functions. If the information is too inaccurate to form a proof that the bug is possible, then nothing is reported.

We were surprised to find elementary errors in so many of the benchmarks, especially in the Linux kernel modules. Our expectation was that such widely used open-source software would not contain such errors, but odd corner cases in less commonly used device drivers are not tested that much. Examining user reports suggests that runtime errors are common

```
void handle_vq(struct c2_dev *c2dev, u32 mq_index)
{
  void *adapter_msg, *reply_msg;
  struct c2wr_hdr *host_msg;
  int err;
  // ...
  host_msg = vq_repbuf_alloc(c2dev);
  // ...
  if (!host_msg) {
    host_msg = &tmp;
    memcpy(host_msg, adapter_msg, sizeof(tmp));
    reply_msg = NULL; // reply_msg explicitly set to NULL
  } else {
    memcpy(host_msg, adapter_msg, reply_vq->msg_size);
    reply_msg = host_msg;
  }
  // ...
  err = c2_errno(reply_msg); // NULL argument not acceptable
  // ...

  if (!err) switch (req->event) {
    case IW_CM_EVENT_ESTABLISHED:
      // ...
  }
}

int c2_errno(void *reply) {
  switch (c2_wr_get_result(reply)) { /* ... */ }
}

unsigned c2_wr_get_result(void *wr) {
  return ((struct c2wr_hdr *) wr)->result;
}
```

Fig. 14. A potential NULL pointer dereference in the amso1100 Linux driver. The `c2_errno` function crashes if passed NULL, yet `handle_vq` can deliberately do so.

with USB devices and network cards, among others. The significant number of such reports we found online suggest that null pointer dereferences, for example, still haunt users despite the apparent maturity of the software.

Not surprisingly, errors often span more than one procedure, often in separate source files that may have been written by different programmers. Of course, these are harder to find and provide motivation for inter-procedural analysis for finding bugs. For example, the procedure in Fig. 14 from the amso1100 Linux device driver can trigger a null pointer dereference.

Fig. 14 would be correct if `c2_errno` could handle a NULL argument (the `reply_msg` variable is explicitly assigned NULL in the *then* branch of the *if* statement). However `c2_errno` immediately calls `c2_wr_get_result`, which unconditionally dereferences its argument `wr`. This error manifests itself when `host_msg` receives a null value from `vq_repbuf_alloc`, which means it failed to allocate a buffer. The programmer did not intend to crash due to a null pointer dereference if this happened: the fact that the value of `err` is tested indicates that the program was expected to continue.

This error can be caught because the summarized information for `c2_errno` contains the fact that its argument `reply` is unconditionally fetched (after incorporating the summarized information from `c2_wr_get_result`). For that to happen, the analysis needs some minimum information about program conditions. Also, statement order information is needed for `handle_vq`, as well as the ability to distinguish *then* and *else* branches. Otherwise, the analysis cannot figure out the relationship between statements `reply_msg = NULL` and `reply_msg = host_msg`. Note `c2_wr_get_result` is not faulty because it does not check for `wr` before dereferencing it; the fault comes from `c2_errno` passing in an illegal input. Interprocedural analysis is necessary here since the statements leading to the error span several procedures.

Balsa also has a potential null pointer dereference in an exception case, shown in Fig. 15. Perhaps the *if* statement is not supposed to fail. If it does, however, `use_from` is assigned NULL, and then later dereferenced by `use_from->address`. Being able to distinguish among the mutually exclusive branches of the *if*, as well as modeling some statement order, makes this error not hard to find.

Perhaps one of the most interesting errors we have reported occurs in the USB module of the Linux kernel, and it involves order, fields, and variable aliasing. Fig. 16 shows a simplified version of the code. The potential error is a null pointer dereference of `urb->dev->tt` in the last statement. This can happen because `tt` and `urb->dev->tt` may be aliased due to statement `tt = urb->dev->tt` at the top, and failing at the null check "`tt ?`" suggests that NULL is meant to be allowed. In response to our bug report, the developers wrote

> Looks to me like this is a longstanding bug in the root hub TT support. See if this patch makes that work better.

```
static void handle_mdn_request(LibBalsaMessage *message)
{
  gboolean suspicious, found;
  const InternetAddressList *use_from;
  // ...
  if (message->headers->reply_to)
    use_from = message->headers->reply_to;
  else if (message->headers->from)
    use_from = message->headers->from;
  else if (message->sender)
    use_from = message->sender;
  else
    use_from = NULL; // SETTING TO NULL

  // ...
  suspicious =
     !libbalsa_ia_rfc2821_equal(message->headers->dispnotify_to->address,
                          use_from->address); // DEREFERENCE

  if (!suspicious) {
    // ...
  }
  // ...
}
```

Fig. 15. A potential null pointer dereference (from Balsa).

```
struct ehci_qh *qh_make(...)
{
  // ...
  if (type == PIPE_INTERRUPT) {
    // ...
  } else {
    struct usb_tt *tt = urb->dev->tt;
    int         think_time;
    // ...
    think_time = tt ? tt->think_time : 0;
  }

  // ...

  switch (urb->dev->speed) {
    // ...
    case USB_SPEED_FULL:
    if (!ehci_is_TDI(ehci) || ...)
      info2 |= urb->dev->tt->hub->devnum << 16;
// DEREFERENCE urb->dev->tt
  // ...
}
```

Fig. 16. An error in the Linux USB module.

> It looks like this is an ARC-derived core, and no root hub TT has been set up. Moreover, it looks like even the original patch adding root hub TT support (only for the PCI based devboard/FPGA) didn't actually set up such a TT . . . so this bug has been around for a very long time.

Although this is not a particularly intricate error, the developers had failed to find it manually or during automated testing. This reinforces our intuition that simple enhancements to static analysis techniques can go a long way towards making software less faulty.

Returning from a function when a global pointer refers to a deallocated block of memory is another common type of error. While such a global variable is harmless if it is overwritten before it is used, it can lead to a hard-to-find error. In the *match* function from the *pcre* benchmark, the statement `md->recursive = &new_recursive` assigns the address of `new_recursive`, a local variable, to `md->recursive`, an outside pointer. Several lines later, the function returns without modifying that pointer. Similarly, the Linux kernel's `start_unregistering` function executes `p->unregistering = &wait` to assign the address of a local variable to outside pointer `p->unregistering` and then returns.

Table 3 indicates that our tool does report false positives (i.e., warns about situations that are not actually bugs). This was usually due to a lack of information about the semantics of a callee function, or errors that would exist only if a loop executes zero times, which is often impossible.

11. Conclusions

Pointer analysis is still an active area of research [47–49]. The number of papers in the subject is no fewer than a hundred, and several Ph.D. theses have been published exclusively on pointer analysis. Being a critical component for most software analysis tools, there is a lot of interest in the problem and many researchers are trying to develop ingenious solutions. Being a very difficult problem, it is unlikely that any of these attempts will solve it in general; instead, we believe that the idea is to tailor the analysis to a particular application.

We have learned that a relevant increment in analysis precision should not require a large decrease in efficiency, and that interesting trade-offs can be obtained by looking at pointer analysis in different ways. For example, the execution-insensitive nature of some dataflow analysis algorithms makes software analysis tools to grossly underreport errors. Providing some minimal order is cheap and can assist with information the tool can rely upon when facing uncertainty about a procedure's execution. Loops may pose a challenge to that objective, and there are basically two ways of dealing with them. Either penalize the entire function by enclosing it on a big cycle, or take the loops apart and analyze them separately, converting the rest of the procedure's code into an acyclic representation. In general, algorithms for acyclic structures are simpler and more efficient, and can yield better results. Our framework is built on top of such kind of representation, and therefore it is simple and efficient. We also find it to be elegant since the definition of a single inference that can be refined in different ways covers a large set of pointer analysis algorithms. The ability to evaluate a myriad of such variations is one of the strengths of our technique, perhaps more important than the specific results of a particular implementation itself.

Like us, Burns and Chandra [50] attempt to characterize multiple abstractions for pointer analysis. They also start with a single representation of program behavior (in their case, labeled transition systems) and use it differently to capture different abstractions for pointer analysis (in their case, they characterize different abstractions as sets of program transformations). We have not attempted a precise technical comparison of our two techniques, although we suspect each approach would have something to learn from the other. However, it does not appear that they have applied their technique to a real-world tool.

We have not tried to make our analysis demand-driven, but we find that our Assign-Fetch Graph representation could be adapted for such purpose since it would not require either constructing or intersecting points-to sets like most existing demand-driven approaches. The structure of the AFG could be explored to derive alias edges on the fly, and only for those operations that are relevant for a given query. This would resemble the work of Sridharan et al. [51], in which a demand-driven points-to analysis for Java is proposed. A key to their approach is to match loads and stores on the same class field through "match" edges. Their technique does not apply for C, however, where one can explicitly take the address of a variable, the address of a field, or dereference any pointer (not only fields like Java).

Indeed, the Assign-Fetch Graph representation has a number of natural features that makes it a very attractive abstraction. Because it models pointer assignments within a function instead of points-to relations, it is used to answer MOD and other side-effect questions directly. In addition, fetch edges in a procedure's summary, specially if annotated with some form of conditions, can provide information about which variables are dereferenced. This plus the ability to summarize a function for any possible calling context makes it ideal for bottom-up, modular analysis. It is our expectation that the AFG will be extended by others in some currently unknown but certainly interesting way.

References

[1] Linux Kernel Project, http://www.kernel.org.
[2] M. Hicks, GM to software vendors: Cut the complexity, http://www.eweek.com/c/a/Enterprise-Applications/GM-to-Software-Vendors-Cut-the-Complexity/(Oct. 2004).
[3] R. Lockridge, Will bugs scare off users of new Windows 2000?, http://archives.cnn.com/2000/TECH/computing/02/17/windows.2000 (Feb. 2000).
[4] S.C. Misra, V.C. Bhavsar, Relationships between selected software measures and latent bug-density: Guidelines for improving quality, in: Proceedings of the International Conference on Computational Science and its Applications, ICCSA, in: Lecture Notes in Computer Science, vol. 2667, Springer, Montreal, Canada, 2003, pp. 724–732.
[5] G. Myers, The Art of Software Testing, 2nd ed., John Wiley and Sons, 2004.
[6] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.
[7] P. Godefroid, Model checking for programming languages using Verisoft, in: Proceedings of Principles of Programming Languages, POPL, Paris, France, 1997, pp. 174–186. http://citeseer.ist.psu.edu/godefroid97model.html.
[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A static analyzer for large safety-critical software, in: Proceedings of Program Language Design and Implementation, PLDI, 2003, pp. 196–207.
[9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang, Cyclone: A safe dialect of C, in: Proceedings of the USENIX Annual Technical Conference, Monterey, California, 2002, pp. 275–288.
[10] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Proceedings of Program Language Design and Implementation, PLDI, Snowbird, Utah, 2001, pp. 59–69.
[11] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, in: Proceedings of Program Language Design and Implementation, PLDI, Berlin, Germany, 2002, pp. 234–245.
[12] D. Engler, M. Musuvathi, Static analysis versus model checking for bug finding, in: Proceedings of Verification, Model Checking and Abstract Interpretation, VMCAI, in: Lecture Notes in Computer Science, vol. 2937, Springer, Venice, Italy, 2004.

[13] D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, in: Proceedings of the Network and Distributed System Security Symposium, NDSS, San Diego, CA, 2000, pp. 3–17.
[14] M. Buss, Summary-based pointer analysis framework for modular bug finding, Ph.D. Thesis, Columbia University, New York, USA, cUCS–013–08 (Feb. 2008).
[15] M. Buss, D. Brand, V. Sreedhar, S.A. Edwards, Flexible pointer analysis using assign-fetch graphs, in: Proceedings of the Symposium on Applied Computing, SAC, Fortaleza, Ceará, Brazil, 2008, pp. 234–239. http://doi.acm.org/10.1145/1363686.1363746.
[16] M. Emami, R. Ghiya, L.J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: Proceedings of Program Language Design and Implementation, PLDI, Orlando, FL, 1994, pp. 242–256.
[17] W. Landi, B. Ryder, A safe approximate algorithm for interprocedural pointer aliasing, in: Proceedings of Program Language Design and Implementation, PLDI, San Francisco, CA, 1992, pp. 235–248.
[18] B. Blanchet, Escape analysis for object oriented languages: Application to Java, in: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Denver, CO, 1999, pp. 20–34.
[19] S. Cherem, R. Rugina, A practical escape and effect analysis for building lightweight method summaries, in: Proceedings of Compiler Construction, CC, in: Lecture Notes in Computer Science, vol. 4420, Springer, Braga, Portugal, 2007, pp. 172–186.
[20] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, S. Midkiff, Escape analysis for Java, in: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Denver, CO, 1999, pp. 1–19.
[21] M. Sagiv, T. Reps, R. Wilhelm, Solving shape-analysis problems in languages with destructive updating, in: Proceedings of Principles of Programming Languages, POPL, St. Petersburg Beach, FL, 1996, pp. 16–31.
[22] R. Wilhelm, S. Sagiv, T.W. Reps, Shape analysis, in: Proceedings of Compiler Construction, CC, in: Lecture Notes in Computer Science, vol. 2027, Springer, Berlin, Germany, 2000, pp. 1–17.
[23] M. Burke, An interval-based approach to exhaustive and incremental interprocedural data-flow analysis, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 341–395.
[24] D. Chase, M. Wegman, F. Zadek, Analysis of pointers and structures, in: Proceedings of Program Language Design and Implementation, PLDI, White Plains, New York, 1990, pp. 296–310.
[25] J. Choi, M. Burke, P. Carini, Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects, in: Proceedings of Principles of Programming Languages, POPL, Charleston, SC, 1993, pp. 232–245.
[26] A.V. Aho, M. Lam, R. Sethi, J.D. Ullman, Compilers, Principles, Techniques, and Tools, 2nd ed., Addison-Wesley, Reading, MA, 2006.
[27] L.O. Andersen, Program analysis and specialization for the C programming language, Ph.D. Thesis, DIKU, University of Copenhagen (1994).
[28] J. Banning, An efficient way to find the side-effects of procedure calls and the aliases of variables, in: Proceedings of Principles of Programming Languages, POPL, 1979, pp. 29–41.
[29] M. Burke, P. Carini, J. Choi, M. Hind, Flow-insensitive interprocedural alias analysis in the presence of pointers, in: Proceedings of the Workshop on Languages and Compilers for Parallel Computing, in: Lecture Notes in Computer Science, vol. 1473, Springer, 1995, pp. 234–250.
[30] K. Cooper, K. Kennedy, Inter-procedural side-effect analysis in linear time, in: Proceedings of Program Language Design and Implementation, PLDI, Atlanta, GA, 1988, pp. 487–506.
[31] M. Das, Unification-based pointer analysis with directional assignments, in: Proceedings of Program Language Design and Implementation, PLDI, Vancouver, BC, Canada, 2000, pp. 35–46. http://research.microsoft.com/manuvir/homepage.html.
[32] B. Steensgaard, Points-to analysis in almost linear time, in: Proceedings of Principles of Programming Languages, POPL, St. Petersburg Beach, FL, 1996, pp. 32–41. http://research.microsoft.com/~rusa/papers.html.
[33] M. Shapiro, S. Horwitz, Fast and accurate flow-insensitive points-to analysis, in: Proceedings of Principles of Programming Languages, POPL, Paris, France, 1997, pp. 1–14.
[34] S. Zhang, B. Ryder, W. Landi, Program decomposition for pointer aliasing: A step toward practical analyses, in: Proceedings of Foundations of Software Engineering, FSE, San Francisco, CA, 1996, pp. 81–92.
[35] M. Hind, M. Burke, P. Carini, J.-D. Choi, Interprocedural pointer alias analysis, ACM Transactions on Programming Languages and Systems 21 (4) (1999) 848–894.
[36] M. Hind, A. Pioli, Assessing the effects of flow-sensitivity on pointer alias analyses, in: Proceedings of the Static Analysis Symposium, SAS, in: Lecture Notes in Computer Science, vol. 1503, Springer, Pisa, Italy, 1998, pp. 57–81.
[37] P. Stocks, B. Ryder, W. Landi, S. Zhang, Comparing flow and context sensitivity on the modification-side-effects problem, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA, Clearwater Beach, FL, 1998, pp. 21–31.
[38] S. Zhang, B. Ryder, W. Landi, Experiments with combined analysis for pointer aliasing, in: Proceedings of the Workshop on Program Analysis for Software Tools and Engineering, PASTE, Montreal, Quebec, Canada, 1998, pp. 11–18.
[39] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999, http://www.daimi.au.dk/~hrn/PPA/ppa.html.
[40] R.P. Wilson, M.S. Lam, Efficient context-sensitive pointer analysis for C programs, in: Proceedings of Program Language Design and Implementation, PLDI, La Jolla, CA, 1995, pp. 1–12. http://suif.stanford.edu.
[41] D. Brand, M. Buss, V. Sreedhar, Evidence based analysis and inferring preconditions for bug detection, in: Proceedings of the International Conference on Software Maintenance, ICSM, Paris, France, 2007, pp. 44–53.
[42] R. Chatterjee, B. Ryder, W. Landi, Relevant context inference, in: Proceedings of Principles of Programming Languages, POPL, San Antonio, TX, 1999, pp. 133–146.
[43] E.M. Nystrom, H.-S. Kim, W. mei, W. Hwu, Bottom-up and top-down context-sensitive summary-based pointer analysis, in: Proceedings of the Static Analysis Symposium, SAS, Verona, Italy, 2004, pp. 165–180.
[44] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of Principles of Programming Languages (POPL), Los Angeles, CA, 1977, pp. 238–252. http://www.dmi.ens.fr/~cousot.
[45] D. Brand, F. Krohm, Arithmetic reasoning for static analysis of software, Tech. Rep. RC-22905, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598 (Oct. 2003).
[46] D. Brand, A software falsifier, in: Proceedings of Software Reliability Engineering (ISSRE), San Jose, CA, 2000, pp. 174–185.
[47] B. Hardekopf, C. Lin, The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code, in: Proceedings of Program Language Design and Implementation, PLDI, San Diego, CA, USA, 2007, pp. 290–299.
[48] C. Lattner, A. Lenharth, V. Adve, Making context-sensitive points-to analysis with heap cloning practical for the real world, in: Proceedings of Program Language Design and Implementation, PLDI, San Diego, CA, USA, 2007, pp. 278–289.
[49] M. Sridharan, R. Bodik, Refinement-based context-sensitive points-to analysis for Java, in: Proceedings of Program Language Design and Implementation, PLDI, Ottawa, Canada, 2006, pp. 387–400.
[50] G. Bruns, S. Chandra, Searching for points-to analysis, IEEE Transactions on Software Engineering 29 (10) (2003) 883–897.
[51] M. Sridharan, D. Gopan, L. Shan, R. Bodik, Demand-driven points-to analysis for Java, in: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, San Diego, CA, 2005, pp. 59–76.